Software-Defined Radio

Lab 2: Data modulation and transmission

Version 0.4

Anton Blad, Mikael Olofsson, and Emil Björnson

November 16, 2015

1 Introduction

In this lab exercise, you will build a transmitter and a corresponding receiver for simple DBPSK modulated data transmission, framed with a correlation sequence in the beginning. The receiver prints the estimated SNR of the received data, as well as the computed bit error rate. You will then modify the transmitter to include an AWGN channel model and verify that the measured SNR and bit error rate correspond to the theoretical values. You will further modify the system to use DQPSK modulation instead of DBPSK to increase the bit rate.

2 Preparations

To finish the lab exercise on time, the following preparation tasks should be done in advance.

1. Read through the entire document. Pay particular attention to the flowgraphs of the transmitter and the receiver. Many of the blocks at the transmitter have counterparts at the receiver. Identify these pairs of blocks.
2. Determine suitable values of the unspecified parameters that are mentioned in Section 4.2:
   - As a first step, determine the number of samples per second that are sent to the USRP for interpolation and transmission.
   - Select the number of samples per symbol (≥ 4) and the number of data symbols per frame to achieve a useful data rate of 180 kbit/s. (Hint: You can write this as an equation with two unknown parameters and try different solutions. There is more than one feasible solution.)
   - Select the number of required correct bits in the frame correlator, according to the guidelines in Section 4.2.

3 Setup

The following hardware is needed for this lab exercise:

- A computer with USB 2.0.
- A USRP1 with an LFTX/LFRX daughterboard pair at the A side.
- An oscilloscope with two channels.

You need GNU Radio installed. The lab exercise is tested and ensured to work with GNU Radio 3.4.2, and assumes that it is installed in `/usr/local/gnuradio-3.4.2`.

3.1 Initialization

If you are doing this lab in CommSys' research lab:
Log in to one of the lab computers as the user *SDR lab user*. Your lab teacher will give you the password. Your lab computer is prepared with all needed files in the directory

```
/home/sdrlabuser/labfolder/lab2
```

Open a terminal and move into that directory using `cd`. You do not need to worry about making changes to files there, because all changes will be reverted for the next lab occasion.

If you are not following these instructions in CommSys' research lab:
Open a terminal. Create a directory `labfolder` somewhere and cd into it. If you have done Lab 1, you should already have that directory. Download and extract the accompanying lab files by executing the following commands:

```
$ wget http://www.commsys.isy.liu.se/SDR/labs/gr/lab2.tar.gz
$ tar zxf lab2.tar.gz
$ cd lab2
```

All files needed in this lab are now in the directory `lab2` that you just created.

4 Data Transmission

Full synchronization is assumed in many communication courses given at Linköping University and at other universities. That is also the main assumption in the majority of lectures and tutorials in TSKS01 Digital Communication. The reason for that is that synchronization issues are relatively complicated to model and solve, thus it could be the topic of a full course. It is a significant problem in engineering reality, which must be taken into account in the system design. In this lab exercise, we have real signals and there is no artificial synchronization of transmitters and receivers. Instead, our transmitters and receivers must include procedures that achieve an approximate synchronization, which entails frequency correction and bit synchronization.

The transmission consists of a long sequence of symbols. This sequence is divided into frames of a given length. As a first step, we need to know where each frame starts. To enables this, the transmitter prepends each frame by a pseudo-noise sequence that marks the beginning of the frame. The rest of the frame contains data symbols. The receiver knows the pseudo-noise sequence,

and correlates the received symbol stream by that sequence. A large correlation marks—with high probability—that we have found the beginning of a frame. This is frame synchronization. The receiver then continues to detect the whole frame. The next frame follows directly after the previous one.

The absolute phase will be unknown in the receiver, but will essentially be unchanged from symbol to symbol since the hardware has a stable behavior. Therefore, we will use differential phase-shift keying (PSK) in the communication systems in this lab exercise. That simply means that the phase difference between adjacent symbols carry the information. This is described in more detail below. Details about how and why this kind of synchronization works is discussed in the follow-up course TSKS04 Digital Communication Continuation Course.

4.1 Noise-Free Transmission Link

In the following, let M be the size of the modulation scheme used, and let $k = \log_2(M)$ be the number of bits per symbol.

The simple data transmitter is shown in Figure 1. It consists of a data source generating a stream of data vectors that are all equal. Each data vector is framed with a simple correlation sequence at the beginning, then converted to a stream of bits. The stream of bits is packed to a number of bits per sample, corresponding to the chosen modulation, then differentially encoded, mapped to a complex symbol, and then passed through a root-raised-cosine shaping filter. The signal is finally modulated to the carrier frequency by the USRP.

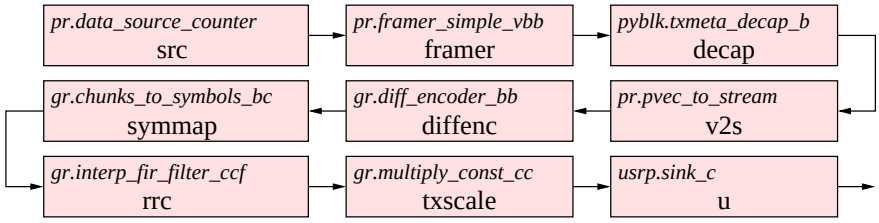

Figure 1: The flowgraph of a transmitter for differential PSK modulated data.

Compare Figure 1 with the stub file `psk_tx.py`. The blocks perform the following operations in order:

src (`pr.data_source_counter`)
: Generate data blocks of known data. The reason that we want to transmit known data is to give us an easy way to check if the received data is correct.
Uses the variable `data_size` in the Python code.

framer (`pr.framer_simple_vbb`)
: Prepend each data block by a pseudo-noise sequence.
Uses the variables `frame_size` and `corrseq` in the Python code.

decap (`pyblk.txmeta_decap_b`)
: Remove meta data header from a packet.
Uses the variable `frame_size` in the Python code.

v2s (`pr.pvec_to_stream`)
: Convert frames to a stream of bits.
Uses the variable `frame_size` in the Python code.

diffenc (`gr.diff_encoder_bb`)
: Differential encoder. $y[n] = (x[n] + y[n-1]) \pmod{M}$. The result are the symbol indices. Uses the variable `mod_order` in the Python code.

symmap (`gr.chunks_to_symbols_bc`)
: This block maps symbol indices onto the actual symbols, which are complex numbers. Uses the variable `constpoints` in the Python code.

rrc (`gr.interp_fir_filter_ccf`)
: An FIR filter that also upsamples (interpolates) the signal. This is a so-called root-raised-cosine filter, which shapes the signal to increases the spectral roll-off of the modulation. Uses the variables `samples_per_symbol` and `rrctaps` in the Python code.

txscale (`gr.multiply_const_cc`)
: Amplify the signal to fit the USRP signal levels.

u (`usrp.sink_c`)
: Send the sampled complex baseband signal to the USRP transmit chain, resulting in the physical output signal. Uses the variable `usrp_interp` in the Python code.

On the receiver side, the structure is shown in Figure 2. The baseband processing includes an automatic gain control, which makes sure that the signal has a desired amplitude level, followed by the `gr_mpsk_receiver_cc` block that does frequency correction and timing recovery. The output of the block contains complex symbol estimates. The symbol estimates are decoded and unpacked to a bit stream. A frame correlator searches for the correlation sequence, and frame sync samples the frame when the correlation sequence is found. The correlation sequence is removed, the header is stripped from the data vectors, and the resulting data stream is fed to a bit error rate computation block.

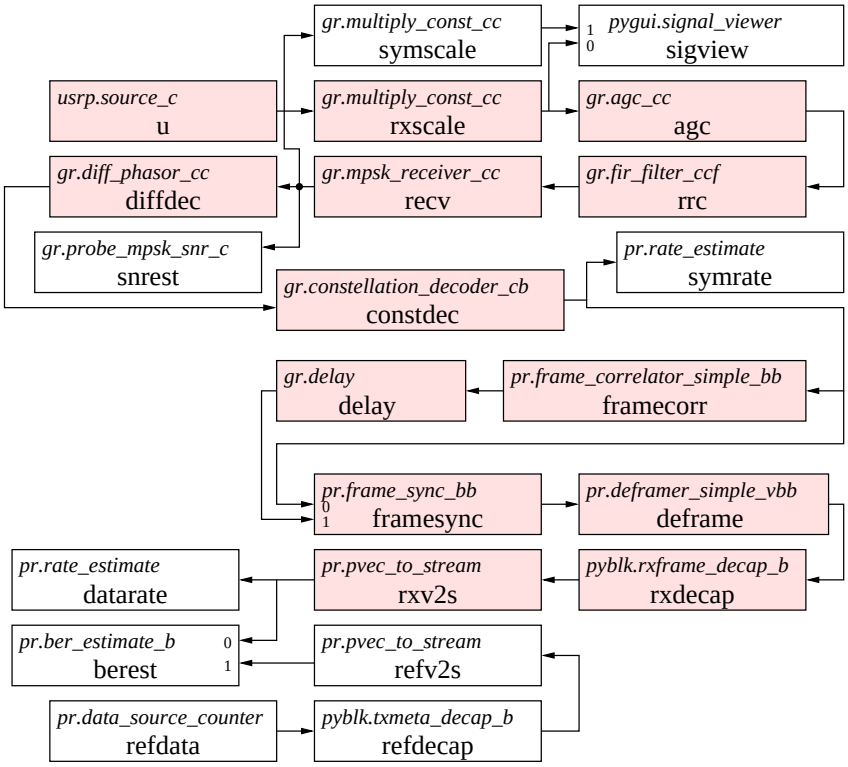

Figure 2: Flowgraph of a simple receiver for differential PSK modulated data.

Compare Figure 2 to the stub file `psk_rx.py`. The pink blocks in Figure 2 constitute the actual signal path. Those blocks perform the following operations in somewhat rough order:

u (`usrp.source_c`)
: Receive the complex baseband signal from the USRP receive chain, which is a sampled and demodulated version of the physical input signal.
Uses the variable `usrp_decim` in the Python code.

rxscale (`gr.multiply_const_cc`)
: An amplifier. Adjust signal levels.

agc (`gr.agc_cc`)
: Automatic gain control. Make sure that the output has the desired amplitude.

rrc (`gr.fir_filter_ccf`)
: Again a root-raised-cosine filter, which is matched filter to the root-raised-cosine filter of the sender. The combined effect is a raised-cosine filter. This block allows for decimation (down-sampling) as well, but that is not used here.
Uses the variable `rrctaps` in the Python code.

recv (`gr.mpsk_receiver_cc`)
: Performs M-PSK detection including carrier synchronization and symbol timing recovery. Uses a Costas loop for phase and frequency recovery and a method called Mueller and Müller for timing recovery. Outputs a sequence of most likely signal points. Both those recovery methods need a bunch of parameters.
Uses the variables `mod_order`, `costas_alpha`, `costas_beta`, `costas_fmin`, `costas_fmax`, `mm_mu`, `mm_gain_mu`, `mm_omega`, `mm_gain_omega` and `mm_omega_rel` in the Python code. Those starting with "`costas_`" are parameters used in the Costas loop, and those starting with "`mm_`" are parameters used in the Mueller and Müller method.

diffdec (`gr.diff_phasor_cc`)
: Differential decoder. Uses the phase difference between two symbols to determine the output symbol:

$$\text{out}[i] = \text{in}[i] \cdot \text{conj}(\text{in}[i-1])$$

Outputs a sequence of complex numbers, whose arguments are differences of arguments of consecutive symbols, i.e., a sequence of M-PSK symbols, where the differential encoding has been removed.

constdec (`gr.constellation_decoder_cb`)
: Maps signal points on bits.
Uses the variable `constpoints` in the Python code.

framecorr (`pr.frame_correlator_simple_bb`)
: Correlates the received stream of bits with a the known pseudo-noise sequence, to find the start of a frame.
Uses the variables `frame_size`, `corrseq` and `ncorr_req` in the Python code.

delay – not explicitly called that in the Python code. (`gr.delay`)
: Delay the signal by one sample.

framesync (`pr.frame_sync_bb`)
: Sample frames when correlation sequence is found.
Uses the variable `frame_size` in the Python code.

deframe (`pr.deframer_simple_vbb`)
: Remove the pseudo-noise sequence from the beginning of the frame, and output only the information bits.
Uses the variables `frame_size` and `corrseq` in the Python code.

rxdecap (`pyblk.rxframe_decap_b`)
: Remove meta data header from a frame.
Uses the variable `data_size` in the Python code.

rxv2s (`pr.pvec_to_stream`)
: Map data to stream of bits.

Uses the variable `data_size` in the Python code.

The white blocks in Figure 2 perform analysis of the received signal and present that in the GUI. In order to provide reference data, a copy of the data generation chain in the transmitter is also included in the receiver. Also, the `fftsink` and `constsink` instances are used to visualize the spectrum and constellation of the received signal. Finally, the symbol estimates are fed to an SNR estimation block. The main loop of the program prints the estimated SNR and computed bit error rate regularly. Those blocks perform the following operations in somewhat rough order:

symscale – not explicitly called that in the Python code. (`gr.multiply_const_cc`)
An amplifier. Adjust signal levels.

sigview (`pygui.signal_viewer`)
Present the signal in the GUI.
Uses the variables `usb_rate` and `samples_per_symbol` in the Python code.

snrest (`gr.probe_mpsk_snr_c`)
Estimate the SNR of an M-PSK signal using the Squared Signal to Noise Variance (SNV) technique, assuming an AWGN channel. This is done by computing the running average of the signal mean and noise variance. This SNR estimator is inaccurate below about 7 dB SNR.

symrate (`pr.rate_estimate`)
Estimate the symbol rate.

datarate (`pr.rate_estimate`)
Estimate the bit rate.

refdata (`pr.data_source_counter`)
Generate a sequence of known data, the same data as the sender uses. The reason for this is so that we can check the error rate by comparing this data to the received data.
Uses the variable `data_size` in the Python code.

refdecap (`pyblk.txmeta_decap_b`)
Remove meta data header from packet.
Uses the variable `data_size` in the Python code.

refv2s (`pr.pvec_to_stream`)
Convert frames to a stream of bits.
Uses the variable `data_size` in the Python code.

berest (`pr.ber_estimate_b`)
Estimate the bit error rate (BER).

4.2 Building a Noise-Free Transmission Link

Use the files `psk_tx.py` and `psk_rx.py` as stubs for building the flowgraphs in Figures 1 and 2, respectively. Design the transmitter and receiver according to the following requirements:

- The frames are sent one after the other, without any silence in between.
- Each frame contains a correlation sequence of 64 bits (defined in the stub files).
- Use differential BPSK modulation for data transmission.
- The usable data rate shall be at least 180 kbit/s. Note that the correlation sequence is not counted into this rate, but is seen as overhead.
- Use a USRP transmitter interpolation rate of 128 and a receiver decimation rate of 64.
- For the timing recovery to work, the number of samples per symbol must be at least 4.

- Use root-raised cosine filters with an excess bandwidth of 0.35.
- Use a carrier frequency of 1 MHz or higher.
- Use the following parameters for the mpsk_receiver block:
  - `costas_alpha = 0.1`
  - `costas_beta = 0.25 * costas_alpha * costas_alpha`
  - `costas_fmin = -0.1`
  - `costas_fmax = 0.1`
  - `mm_mu = 0.5`
  - `mm_gain_mu = 0.01`
  - `mm_omega = samples_per_symbol`
  - `mm_gain_omega = 0.25 * mm_gain_mu * mm_gain_mu`
  - `mm_omega_rel = 0.005`
- Adjust the number of required correct bits in the frame correlator such that the probability of a false positive correlation (assuming random data) is at most 10^{-6}, and the probability of a successful correlation is at least 99.9% for an SNR of 5 dB. You may use Octave or Matlab and the bundled `cseq.m` for this.

To design the root-raised-cosine wave-shaping filters, use the function

`gr.firdes.root_raised_cosine`

This function requires both the sampling frequency and the symbol rate, but depends in reality only on the relation between the two. For a given number of samples per symbol (`samples_per_symbol`), the parameters for the root-raised-cosine filters can be determined according to the following guidelines, for the transmitter:

- `gain = 1.0`
- `sampling_freq = samples_per_symbol`
- `symbol_rate = 1`
- `alpha = 0.35`
- `ntaps = 11 * samples_per_symbol`

For the receiver, use instead the following:

- `gain = samples_per_symbol`
- `sampling_freq = samples_per_symbol`
- `symbol_rate = 1`
- `alpha = 0.35`
- `ntaps = 11 * samples_per_symbol`

Connect the USRP to the host computer as described in Lab 1. Run the transmitter and receiver and verify that the transmission is flawless and that the rate corresponds to the expected one. Next, connect the oscilloscope to the USRP, as in Lab 1, and investigate the shape of the transmitted signal.

4.3 Simulating a Channel

GNU Radio provides complete channel models for both AWGN and different fading channels. However, in this lab exercise, you will build your own channel model using a noise source and an adder. Implement an AWGN channel according to Figure 3, and insert it before the shaping filter in the transmitter. This placement adds the distortion to the symbols rather than the samples,

which would be more practically more appropriate. However, it assures that all the added noise is in the same band as the signal, rather than spread out over the whole spectrum (so that no additional noise-filtering is needed).

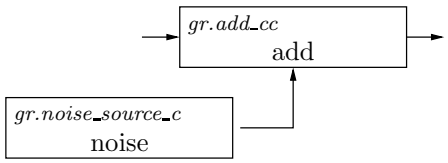

Figure 3: A simple implementation of an AWGN channel.

The parameter of the noise source can be set through the `set_amplitude()` function. This sets the standard deviation if Gaussian noise is specified. In your program, you should be able to specify the desired SNR of the channel (assuming a unit signal energy), and the noise source should be set accordingly.

Your transmitter code already acknowledges the command line command `q` to quit the program. We now want your receiver also to acknowledge the command line command `n SNR`, where `SNR` should be the wanted signal-to-noise ratio expressed as a number in dB.

Adjusting the if-statement near the bottom of the transmitter code to something similar to the following should give you the opportunity to set the SNR via the command line:

```
if cmd[0] == 'q':
        quit = True
elif cmd[0] == 'n':
        SNRdB = eval(cmd[1])
        tb.noise.set_amplitude(10**(-SNRdB/20.0))
else:
        print "Invalid command"
```

You may want to add a suitable print statement above the if statement to inform the user about the changes in the user interface. This assumes that you have created the attribute `noise` of the class as `self.noise`, and it should be a `set_amplitude()` object. Note that it is important that the division is by the float 20.0 and not by the integer 20. Otherwise, you will get into trouble for integer SNR values.

4.4 If Time Permits: Increasing the Data Rate

Double the data rate by changing the transmitter and receiver to use QPSK instead of BPSK. The following can come in handy:

```
j = complex(0,1)
constellation = [1,j,-1,-j]
```

5 Resources

- Online USRP FAQ:
  `https://gnuradio.org/redmine/projects/gnuradio/wiki/FAQ`
- Firas Abbas Hamza, *The USRP under 1.5X Magnifying Lens!*:
  `http://gnuradio.org/redmine/attachments/129`
  Third-party documentation of USRP.
- GNU Radio C++ block documentation:
  `file:///usr/local/gnuradio-3.4.2/share/doc/gnuradio-3.4.2/html/index.html`
- Firas Abbas, *Simple User Manual for Gnuradio*:
  `https://www.csun.edu/~skatz/katzpage/sdr_project/sdr/Simple-Gnuradio-User-Manual-v1.0.pdf`
  Third-party documentation of GNU Radio. This one is a bit out of date, but still useful. A printed copy is available at the lab desk.